The Architecture of newmake

Table of contents

1. Revision history

Date	Description
2006-04-15	Initial version.
2006-04-17	Added the References section.

2. Introduction

The present documents tries to give a description of the involved concepts in newmake. The emphasis is on concepts, not on details. It will describe a large number, but not all of the target. It does not aim at a complete up-to-date descriptions of all targets, their prerequisites, side effects, etc. For this, the reader is referred to the sources, which even contains some comments(!).

Good documentation is not program code translated to English.

The reader is supposed to know the "introductory" document, and to have some experience and understanding of compiling programs in the GNU automake/autoconf-environment.

The Tuxbox build system has grown over a long period of time. The original developers are, with very few exceptions, no longer active within the project. In several cases, quite horrible techniques have been employed. Newmake is an attempt to clean up some of the problems; to solve it as it should have been done the first time. However, I have not went through all components. There are still some fundamental problems inherited from "old make".

Tuxbox uses the GNU automake/autoconf system. Understanding this on the surface is not too hard, however understanding the inner workings, and its customizations is not a trivial undertaking. Here we just mention that from the file `configure.ac` the non-interactive configuration script `configure` is created, while a Makefile is generated from the file `Makefile.am`. For this, some Tuxbox-specific m4-macros are found in the file `acinclude.m4`.

3. Organization of the file systems

3.1. cvsdir

The top level directory that was checked out from CVS (it contains a subdirectory named `cdk`) will be denoted *cvsdir*. (There is no make-variable with that name!)

3.2. cvsdir/cdk

Located as a subdirectory to *cvsdir*. This is the directory where the make-commands are

issued. Corresponds to the make-variable `buildprefix`.

3.3. cdk

The file system `$(hostprefix)`, for example `/tuxbox/cdk`, contains the "Cross Development Kit (CDK)". (We will refer to it as *cdk*.) Contained therein is the cross C and C++ compiler (with support files), as a number of utilities for creating and manipulating programs to be run on the dBox. Also, some programs, built during the tuxbox build, like `mkflfs` are installed here. Include-files for the C++-compiler and stdc++ library are found here. Documentation files for some of the installed component (man- and GNU info-files) are also installed. This directory hierarchy is build during the "bootstrap" build.

3.4. cdkroot

The file hierarchy `$(targetprefix)` (typically `/tuxbox/cdkroot`) is mounted as root file system for a YADD-setup. (We will refer to it as *cdkroot*.) However, it plays some more roles. In the original makefile, images were built by first installing (using the component's makefiles) in *cdkroot*, then selectively copying over selected files to the image file systems. The directly hierarchy is "mainly" built during the cdk build, however, some crucial components (belonging to the C library `glibc`) are installed during the "bootstrap" build.

Through symbolic links, the above described *cdk*-directory depends on *cdkroot*, and secondarily, through links from *cdkroot* to the kernel sources, on the kernel sources in *cvsdir*`/cvs/linux`. In the future, it would be desirable to eliminate this dependency, in my opinion also if this means multiple copies of the same file. "Single sourcing" does not prohibit multiple copies, it means that you know where every copy came from.

Warning:

It is often tempting to delete *cdkroot*, in order to bring the compilation environment back to the state where the cross development environment CDK has been build, but none of the real Tuxbox software. For reasons just described, this will break CDK.

3.5. cdkflash

The file hierarchy `$(flashprefix)` (typically `/tuxbox/cdkflash`, henceforth denoted by *cdkflash*) is a scratch area for building images. It will be described in detail later. It can be deleted when needed/desired without any side effects.

3.6. bootprefix

For the purpose of this article, `bootprefix` is the target location of the yadd kernel and the corresponding `u-boot` boot loader. Typically, this is the base directory for the TFTP service.

4. Organization of the files for Make

There are hundreds of targets in the top level "Makefile". To improve the overview, it has been split in different components. The top level `Makefile.am`, in the current version 1.480.2.21, is shown in the Appendix. This file first defines some high-level targets (in terms of some other targets), then it includes a large number (presently 54) of makefile fragments, defining other targets. The inclusion is preceeded by a comment stating the purpose of the included fragment.

5. Download-unpack-patch-configure-build-install-clean targets

The Tuxbox software needs a number of third-party software. The build mechanism will download the needed software sources on demand. These are stored in the directory *cvsdir*`/cdk/Archive`. (When having several source trees on the disk, it is a good idea to share this directory, to save disk space and downloads.) When "make-ing" the package, the following things occur:

1. The package source code, typically with a `.tar.gz` or `.tar.bz2` extension, is downloaded to the *cvsdir*`/cdk/Archive` directory,
2. It is unpacked into a temporary directory, residing in the `cdk` directory,
3. In some cases, a patch (residing in the directory *cvsdir*`/cdk/Patches` is applied,
4. The package is build using a package specific build command, typically a configure-command, followed by a make-command, possibly with parameters,
5. The package is installed, typically in *cdkroot*, typically with a "make install"-command, possibly with parameters,
6. The build directory is deleted,
7. The successful build is recorded by creating a zero-length file, having the same name as the package, in the directory *cvsdir*`/cdk/.deps`.

It was attempted to have this behavior completely parameterized, using the files `rules-make`, `rules-archive`, `rules-installrules-install-flash`. To this end, `rules-make` defines the package name, version, name of the build directory, name of the current source code distribution file, a command to unpack and possibly to patch. The file `rules-archive` contains a mapping from file names (as given in the `rules-make` file) to download-URLs. Finally, `rules-install` and `rules-install-flash` contain the commands to install the package. HEAD-make, if configured for image building, first consults `rules-install-flash` for the install rules, if not found there, it searches `rules-install`. If not configured for flash images, it only searches `rules-install`. Newmake, for compatibility, first searches `rules-install-flash`, then `rules-install`.

To the implementation: For every such package, `configure.ac` contains the a call to the local autoconfig macro `TUXBOX_RULES_MAKE` (defined in `acinclude.m4`), using the package name as argument. Thus, during executing of `configure`, a few Perl programs are executed, operating on the `rules-*`, thereby defining the shell variables

`DEPENDS_`*package*, `DIR_`*package*, `PREPARE_`*package*, `VERSION_`*package*, `INSTALL_`*package*, `CLEANUP_`*package*. Thus, in `Makefile.am` (or its included parts), constructs like `@VERSION_`*package*`@` can be used; when automake creates `Makefile` out of `Makefile.am`, these will be appropriately substituted.

This has been an interesting, but not completely successful experiment. The "parameterization" of the build has not been a success; every make rule still looks different. To keep used versions in a separate file still is a good idea, however, this can just as well be achieved with the include-mechanism of (auto-)make.

A re-write would be desirable.

6. Three main sets of targets

There are three main categories of targets in the Makefile: Targets for building the cross compilation environment ("CDK"), targets for YADDs, and targets for flash image creation.

6.1. The development environment CDK

The top level target is `bootstrap`. It turns out, that this is nothing but the gcc target. Almost all non-cdk targets depend on this; in the case this dependency is not in the Makefile, it is likely a bug. There are five components required:

`directories`
Sets up a directory skeleton in *cdk* and *cdkroot*.

`binutils`
Installs the GNU binutils, containing programs for creating and manipulating binary files, like the assembler `as` and loader `ld`.

`linuxdir`
Installs the sources for the Linux kernel. This is necessary for building the compiler, since the latter needs include files from the kernel.

`glibc`
The main C library. Since the C compiler needs this, first a "bootstrap compiler" (target `bootstrap_gcc`, a mini C compiler, not needing `glibc`, just intended to compile `glibc`) is first built.

`gcc`
The C cross compiler, in both C and C++-version.

These targets, with the exception of `directories`, are all download-unpack-patch-configure-build-install-clean targets, in the sense above. The rules are all found in the file `make/bootstrap.mk`.

6.2. YADD builds

Useful high-level targets include: `yadd-neutrino`, `yadd-lcars`, `yadd-enigma`,

and yadd-all.

yadd-neutrino
Installs the targets yadd-none, neutrino, as well as the plugins appropriate for Neutrino (targets neutrino-plugins and fx2-plugins).

yadd-micro-neutrino
This target has mainly theoretical interest, to document the minimal usable Neutrino installation.

yadd-enigma
Installs the targets yadd-none, enigma, as well as the plugins appropriate for Enigma (targets enigma-plugins and fx2-plugins).

yadd-lcars
Installs the targets yadd-none, and lcars.

yadd-all
Installs the targets yadd-none, neutrino, enigma, and lcars.

yadd-none
The GUI-independent parts on a working yadd, consisting of bare-os, together with a number of other, non-GUI-based targets (config, tuxbox_tools, procps, ftpd, yadd-ucodes, version).

bare-os
The minimal setup to run Linux on the dBox, allows to login and `ls`. Depends on targets yadd-u-boot, kernel-cdk, driver, yadd-etc, busybox, modutils, tuxinfo.

plugins
Depends on neutrino-plugins, enigma-plugins, and fx2-plugins.

neutrino-plugins
The name is strictly speaking misleading; the target installs plugins usable with any GUI, except for the plugins in target fx2-plugins. Presently, these are tuxmail, tuxtxt, tuxcom, tuxcal, and vncviewer (all of these correspond to their own individual targets). Defined in make/plugins.mk.

enigma-plugins
Plugins that require Enigma. Defined in make/plugins.mk.

fx2-plugins
Plugins that require the fx2-library. Usable by any GUI. Defined in make/plugins.mk.

neutrino
The Neutrino GUI. Defined in make/neutrino.mk.

enigma
The Enigma GUI. Defined in make/enigma.mk.

lcars
The LCARS GUI. Defined in make/lcars.mk.

config
Installs some configuration files, presently cables.xml and satellites.xml. Defined in make/dvb-config.mk.

tuxbox_tools
installs several different tools, most of which are pretty special, some (like `switch`) absolutely essential. Defined in `make/tuxbox_tools.mk`.

procps
A download-unpack-patch-configure-build-install-clean target. Installs the commands `ps` and `top`. Defined in `make/rootutils.mk`.

ftpd
The ftp-daemon. A download-unpack-patch-configure-build-install-clean target. Defined in `make/ftpd.mk`.

yadd-ucodes
Provided that `--with-ucodesdir` was given when configuring, installs that directory's content in the yadd. Defined in `make/ucodes.mk`.

version
Creates the `/.version`-file in the yadd. Defined in `make/version.mk`.

yadd-u-boot
Creates both the "smart" `u-boot`, as the "dumb" `u-boot-yadd`, both in the `$(bootprefix)` directory. The default `u-boot` relies on a DHCP-server (a bootp server will not do) to tell the name of the kernel file, and the location of the NFS-root. Sometimes this is not available, for example when using the Windows dBox manager. For these cases, an alternate u-boot is provided, which, out-of-the-box, has the file name `u-boot-yadd`. This offers less flexibility, having most file names/paths compiled in. Using this u-boot for booting, the file name of the kernel is `kernel-yadd`, and the NFS root will be `yaddroot`. As a side effect, a tool called `mkimage`, needed for building some images, will be installed in *cdk*`/bin`. (This can also be achieved by calling the target `$(hostprefix)/bin/mkimage` directly). Defined in `make/u-boot.mk`.

kernel-cdk
Creates and installs the Linux kernel, using the path name `$(bootprefix)/kernel-cdk`. Also installs the kernel and a map file in *cdkroot*`/boot`. Defined in `make/linuxkernel.mk`.

driver
Compiles and installs device drivers, corresponding to the kernel. Defined in `make/linuxkernel.mk`.

yadd-etc
Installs the content of the `etc` directory. Defined in `make/etc.mk`.

busybox
Configures the busybox for usage with yadd, compiles and installs it. This is an download-unpack-patch-configure-build-install-clean target. Defined in `make/busybox.mk`.

modutils
Installs some utilities for manipulating loadable kernel modules, e.g. modprobe. This is an download-unpack-patch-configure-build-install-clean

target. Defined in make/rootutils.mk.

tuxinfo
Installs the crucial tuxinfo program. This target is actually a subset of the target tuxbox_tools. Defined in make/tuxbox_tools.mk.

camd2
Installs the camd2 program. This target is actually a subset of the target tuxbox_tools. Defined in make/tuxbox_tools.mk.

6.3. Flash images

The high-level flash targets flash-[neutrino,enigma,all]-[cramfs,squashfs,jffs2]-[1x,2x,all] were introduced in the introductory article. Here we will describe the image building process in more detail. In the sequel, *gui* will denote either neutrino or enigma, *filesystem* will denote either cramfs, squashfs, or jffs2 (the file system of the root partition), while img*X*x will denote either img1x or img2x.

Fundamental for the creation of a *gui-filesystem*.imgXx-image are the three directory hierarchies $(flashroot)/root (containing parts not depending on the root file system type or the GUI), $(flashroot)/root-*filesystem* (containing parts depending on the root file system, in particular the kernel and the drivers), $(flashroot)/root-*gui* (containing the GUI component). There are no "short" make targets for these directories, however they are all, with their full path names, make targets. The flashable directories (root-*gui-filesystem* and var-*gui*) are created by copying the contents of the previously mentioned three directories into one, performing the library reduction and finally installing some additional components, for example by calling appropriate "make install"-commands in the directory *cvsdir*/cdk/root.

From the flashable directories, partition image files are created using the commands $(MKJFFS2), $(MKCRAMFS), and $(MKSQUASHFS). The expansion of these commands will be determined during the configuration run.

Finally, the partition images are combined with a suitable u-boot bootloader, packed in a flfs-partition image, to a full images using the flashmanage.pl program, or, in the case of a jffs2-image, simply concatenated together.

The major targets are listed next.

flash-*gui*.img*X*x
This has as only prerequisite the file $(flashprefix)/*gui*.img*X*x. Defined in the file make/flash-expand-targets.mk.

$(flashprefix)/*gui-filesystem*.img*X*x.
The partition images (root-*gui.filesystem*, possibly var-*gui*..jffs2, and *filesystem*.flfs*X*x) are combined to a full image. Defined in make/fullimages.mk.

$(flashprefix)/root-*gui.filesystem*

The partition image is created from the flashable directory `$(flashprefix)/root-gui-filesystem`. Defined in `make/partition-images.mk`.

`$(flashprefix)/var-gui.jffs2`

The partition image is created from the flashable directory `$(flashprefix)/var-gui`. Defined in `make/partition-images.mk`.

`$(flashprefix)/filesystem.flfsXx`

The appropriate u-boot bootloader is build, and packed into the flfs-partition image, using the program `mkflfs` (code residing in *cvsdir*`/hostapps/mkflfs`), which may be build when needed. Defined in `make/partition-images.mk`.

`$(flashprefix)/root-gui-filesystem`

The contents of the directories `root`, `root-filesystem`, and `root-gui` are first copied together into this directory, library reduction is performed by "making" the target `$(flashprefix)/root-gui-filesystem/lib/ld.so.1`, bootlogos are installed (if applicable), some additional files are installed, for example by calling "make install" in the directory *cvsdir*/cdk/root.

`$(flashprefix)/var-gui`

The contents of the directories `root/var` and `root-gui/var` are first copied together into this directory. Bootlogos are installed (if applicable), some additional files are installed, for example by calling "make install" in the directory *cvsdir*/cdk/root.

`$(flashprefix)/root-gui-filesystem/lib/ld.so.1`

Due to the limited flash memory of the dBox (8 MiB), it is not feasible just to include all possible shared libraries in the image. This step, called "library reduction", makes sure that only actually needed libraries are included, and that they are reduced in size as much as possible. Executable files and shared libraries are stripped (= symbols removed) to reduce their size. The necessary libraries are gathered together, mainly from *cdkroot*, as needed. The work is carried out by the mklibs program. Unneeded files are deleted. The file `ld.so.1` is the runtime loader, and serves as a "marker file" for make, in that the make target has many more "side effects" than just creating this file. The target is defined in the file `make/reduce-libs.mk`.

`$(flashprefix)/root-gui`

"Installation" of the corresponding GUI. There is a synonym (target having this as only prerequisite, and no actions) `flash-gui`. Defined in `make/neutrino.mk` and `make/enigma.mk`.

`$(flashprefix)/root-filesystem`

Installation of kernel and drivers for an image having *filesystem* as its root file system type. Defined in `make/flashroot-fs.mk`.

`$(flashprefix)/root`

Essentially, all components, which do not depend either of the GUI, nor of the file system type of the root file system are installed here. Do not confuse with

non-GUI components; for example plugins usable by any GUI are also installed here. This target does not depend on very much, instead it calls make recursively, to install required components. We do not describe this in further detail; the interested reader is referred to the file `make/flashroot.mk`. Many of the targets there, for example `flash-ftpd` are the flash versions of the yadd targets described above, of course then without the `flash-` prefix.

7. Odds and Ends

7.1. GNU Make "Order-Only Prerequisites"

GNU Make 3.80 introduced a facility, the "order-only prerequisites", that I have found indispensible for newmake. (For this reason, GNU make 3.80 is required for newmake, while HEAD-make is satisfies with 3.79.) Since this feature is not very well known, the name is badly chosen, and the description in the manual not very enlightening, I explain it in detail next.

Consider the following Makefile:

```
buildir=/home/me/somewhere
sourcedir=/home/me/somewhereelse

$(builddir)/prog: $(sourcedir)/prog.c $(builddir)
        $(CC) -o /tmp/foobar $<
        sleep 1
        mv /tmp/foobar $@

$(builddir):
        mkdir $@
```

The intention is to create the build directory (if required), and then to compile `prog` in it. This also works. However, the next time `make` is issued, the compilation is redone! Why? Since the target depends on `$(builddir)`, which gets its timestamp set by the `mv`-command, the target is older than its prerequisite (`$(builddir)`), and thus is scheduled for re-making! (`sleep 1` is inserted in the example to guarantee that the file system dates from prog and the directory differs; it is hard to make a really simple realistic example.) This is most likely not what the Makefile-author had in mind. What is needed is a way to say: $(builddir) need exist, but, as long as it exists, its date should never be taken into account. This is exactly the "order-only" (better would be "existence-only") prerequisite does. Order-only prerequisites are separated from normal prerequisites using the bar "`|`". Thus, changing the first prerequisite line to:

```
$(builddir)/prog: $(sourcedir)/prog.c | $(builddir)
```

achieves the effect the author of the original makefile wanted.

7.2. Maneuvering within the make sources

Emacs users can maneuver quite comfortable within the make sources. For this, first issue the shell command "make TAGS" in the directory *cvsdir*`/cdk`, thereby creating a so-called TAGS-file. Now it will be possible to use the command `find-tag` (normally bound do M-.) to directly jump to the file that defines a particular target. Also other editors have similar tags-support.

8. References

- The GNU Make manual, online version. The only Make book you need.
- The Autoconf manual, online version.
- The Automake manual, online version (slightly outdated version).
- The GNU Coding standards. There is *good stuff* in here. Unfortunately, not observed by the Tuxbox project.
- GNU Autoconf, Automake, and Libtool by Gary V. Vaughan, Ben Elliston, Tom Tromey and Ian Lance Taylor. I just discovered it, and have not read this (yet), however, the table of contents looks very promising.

9. Appendix. Top level Makefile.am.

```
## Makefile for Tuxbox

all:
        @echo "You probably do not want to build all possible targets."
        @echo "Sensible targets are, e.g. yadd-enigma or
flash-neutrino-jffs2-2x."
        @echo "If you REALLY want to build everything, then \"make
everything\""

if TARGETRULESET_FLASH
everything: yadd-all flash-all-all-all serversupport extra
else
everything: yadd-all extra serversupport
endif


#####################################################################
# High-level yadd targets

bare-os: yadd-u-boot kernel-cdk driver yadd-etc busybox modutils tuxinfo
        @TUXBOX_YADD_CUSTOMIZE@

yadd-none: bare-os config tuxbox_tools procps ftpd yadd-ucodes version
        @TUXBOX_YADD_CUSTOMIZE@

yadd-micro-neutrino: bare-os config yadd-ucodes camd2 switch neutrino
        @TUXBOX_YADD_CUSTOMIZE@

yadd-neutrino: yadd-none neutrino-plugins fx2-plugins neutrino
        @TUXBOX_YADD_CUSTOMIZE@

yadd-enigma: yadd-none enigma-plugins fx2-plugins enigma
        @TUXBOX_YADD_CUSTOMIZE@
```

```
yadd-lcars: yadd-none lcars
	@TUXBOX_YADD_CUSTOMIZE@

yadd-all: yadd-none plugins neutrino enigma lcars
	@TUXBOX_YADD_CUSTOMIZE@

extra: libs libs_optional contrib_apps fun dvb_apps root_optional udev
devel bash

# Set up some default values (used only by serversetup).
include make/defaultvalues.mk

# Set up the build environment
include make/buildenv.mk

# Set up the cross compilation enironment, including linux kernel
# source and directory structure
include make/bootstrap.mk

# The automounter (optional)
include make/automount.mk

# The busybox (implements most standard Unix commands, like ls,...)
include make/busybox.mk

# Populate the etc directory in YADD
include make/etc.mk

# The ftpd
include make/ftpd.mk

# Some core tools (important and less important)
include make/rootutils.mk

# A number of libraries, some of which necessary for neutrino or enigma
include make/contrib-libs.mk

# Some non-GUI applications, none of which are essential
include make/contrib-apps.mk

# Tools (debugger etc) for the Tuxbox developer
include make/development-tools.mk

# The kaffe java-implementation (nonessential, presently does not build)
include make/java-stuff.mk

# Gaming platforms (gnuboy scummvm sdldoom)
include make/fun.mk

# Nonessential DVB application
include make/dvb-apps.mk

# Bluetooth (nonessential)
include make/bluetooth.mk

# FUSE and djmount for uPnP support (non-essential)
include make/upnp.mk

# The u-boot boot loader
```

```
include make/u-boot.mk

# Build kernel and its drivers
include make/linuxkernel.mk

# Install dvb configuration files (cables.xml & satellites.xml)
include make/dvb-config.mk

# dvbsnoop is a tool for analyzing dvb streams (non-essential)
include make/dvbsnoop.mk

# A nonessential library
include make/libdvb++.mk

# The zapit daemon
include make/zapit.mk

# More dvb tools, of which only streampes is installed per default
include make/dvb_tools.mk

# More misc libs, mostly nonessential
include make/misc_libs.mk

# Misc tools, not essential
include make/misc_tools.mk

# Enigma GUI
include make/enigma.mk

# nonessential entertainment, like "screensavers" for the lcd display
include make/funstuff.mk

# The LCARS GUI
include make/lcars.mk

# LCD tools
include make/lcd.mk

# Essential, and some less essential, libraries
include make/tuxbox_libs.mk

# A small, but absolutely essential library
include make/libtuxbox.mk

# The Neutrino GUI
include make/neutrino.mk

# Plugins
include make/plugins.mk

# Some small command line tools, several of which are essential
include make/tuxbox_tools.mk

# Application that run on the build host
include make/hostapps.mk

# Generate some support files for a YADD- or flashing-server
include make/serversupport.mk
```

```
# Optionally install ucodes in the image
include make/ucodes.mk

# Generate a /.version file in the image
include make/version.mk

if TARGETRULESET_FLASH
# High-level flash targets are:

# flash-[neutrino,enigma,all]-[cramfs,squashfs,jffs2,all]-[1x,2x,all]
# Expand all flash targets containg the word "all"
include make/flash-expand-targets.mk

# Create complete images ("without BN bootloader")
include make/fullimages.mk

# Create images of the root and var file systems
include make/partition-images.mk

# Create root and var filesystems, ready for image creation
include make/flashable-dirs.mk

# Strip libraries of symbols not needed.
include make/reduce-libs.mk

# Create the root file systems for jffs2-only, cramfs, and squashfs
# images (containing kernel but not GUI)
include make/flashroot-fs.mk

# Create the root file system, without kernel and GUI
include make/flashroot.mk

# The streampes stuff
include make/flash-streampes.mk

# Build distribution lists in neutrino internet update format
include make/distribution-lists.mk

# /etc/cramfs.urls contains URLs for update lists
include make/cramfs.urls.mk
endif

# Files not to be deleted, even though they are intermediate products
include make/precious.mk

# "Phony" make targets
include make/phony.mk

# Create the TAGS file
include make/tags.mk

# A number of cleaning targets
include make/cleantargets.mk

# Target for building source distributions (hardly used these days of
CVS :-)
include make/disttargets.mk

# Give the user rope to hang himself :-).  (Note: read from the
```

```
# generated Makefile during make run, automake or configure does not
# see it.)
-include ./Makefile.local
```